Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs

Amadeo Casas,[1] Manuel Carro,[2] and Manuel V. Hermenegildo[1,2]

{amadeo, herme}@cs.unm.edu
{mcarro, herme}@fi.upm.es

[1] Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA.
[2] School of Computer Science, Universidad Politécnica de Madrid, Spain.

Abstract. We present two new algorithms which perform automatic parallelization via source-to-source transformations. The objective is to exploit goal-level, *unrestricted* independent and-parallelism. The proposed algorithms use as targets new parallel execution primitives which are simpler and more flexible than the well-known `&/2` parallel operator. This makes it possible to generate better parallel expressions by exposing more potential parallelism among the literals of a clause than is possible with `&/2`. The difference between the two algorithms stems from whether the order of the solutions obtained is preserved or not. We also report on a preliminary evaluation of an implementation of our approach. We compare the performance obtained to that of previous annotation algorithms and show that relevant improvements can be obtained.

Keywords: Logic Programming, Automatic Parallelization, And-Parallelism, Program Transformation.

1 Introduction

Parallelism capabilities are becoming ubiquitous thanks to the widespread use of multi-core processors. Indeed, most laptops on the market contain two cores (capable of running up to four threads simultaneously) and single-chip, 8-core servers are now in widespread use. Furthermore, the trend is that the number of on-chip cores will double with each processor generation. In this context, being able to exploit such parallel execution capabilities in programs as easily as possible becomes more and more a necessity. However, it is well-known [17] that parallelizing programs is a hard challenge. This has renewed interest in language-related designs and tools which can simplify the task of producing parallel programs.

The comparatively higher level of abstraction of declarative languages and, among them, logic programming languages, allows writing programs which are closer to the specification of the solution. Besides, there is often more freedom in the implementation of different operational semantics which respect the declarative semantics. In particular, the notion of control in declarative languages frequently allows for more flexibility to arrange the evaluation order of some

operations, including executing them in parallel if deemed convenient, without affecting the semantics of the original program. Additionally, the cleaner declarative semantics makes it possible to automatically detect more accurately any lack of dependencies among operations and hence to exploit opportunities for parallelism more easily than in imperative languages. At the same time, in most other respects in the case of logic programs the presence of dynamic data structures with "declarative pointers" (logical variables), irregular computations, or complex control makes the parallelization of logic programs a particularly interesting case that allows tackling the more complex parallelization-related challenges in a formally simple and well-understood context [11].

Because of this potential, automatic parallelization has received significant attention in logic programming [10], where two main forms of parallelism have been studied. *Or-parallelism* is exploited when the alternatives created by nondeterministic goals are explored simultaneously. Some relevant or-parallelism systems are Aurora [20] and MUSE [1]. *And-parallelism* aims at executing simultaneously (conjunctive) goals in clauses or in the resolvent. Examples of systems that have exploited and-parallelism are DDAS [25] and &-Prolog [12]. Additionally, some systems such as ACE [9], AKL [16], and Andorra [24] exploit certain combinations of both and- and or-parallelism. While or-parallelism can only obtain speedups when there is search involved, and-parallelism can be used in more algorithmic schemes, with divide-and-conquer and map-style algorithms being classic representatives. In this paper, we concentrate on and-parallelism.

A correct parallelization has been defined as one that preserves during and-parallel execution some key properties, typically correctness and no-slowdown [14]. The preservation of these properties is ensured by executing in parallel goals which meet some notion of *independence*, meaning that the goals to be executed in parallel do not interfere with each other in some particular sense. This can include for example absence of competition for binding variables plus other considerations such as, e.g., absence of side effects. For simplicity, in the rest of the paper we will assume that we are only dealing with side-effect free program sections. Note however that this does not affect the generality of our presentation, as we deal with dependencies in a generic way.

One of the best understood sufficient conditions for ensuring that goals meet the efficiency and correctness criteria for parallelization is *strict independence* [14], which entails the absence of shared variables at runtime between any two goals being parallelized. It should be noted that some proposals exploit and-parallelism between goals which do not meet this condition, but on which other restrictions are imposed which also ensure no-slowdown and correctness. Examples of such restrictions are determinism and non-failure [14] (determinism is exploited for example in [24]) and absence of conflicts due to the binding of shared variables (as in *non-strict* independent and-parallelism [14]). Another interesting issue is at what level of granularity the notion of independence is applied: at the goal level, at the binding level, etc. Our work in this paper will focus on *goal-level* (strict and non-strict) independent and-parallelism.

One particularly successful approach to automatically parallelizing a logic program uses three different stages [15, 2, 10]. The first one detects data (and control) dependencies between pairs of literals in the original program. A dependency graph (see Figure 1 as an example) is built to capture this information. Nodes in the graph correspond to literals in the body of the clause and edges represent dependencies between them. Edges are labeled with the associated dependency conditions (which may be trivially *true* or *false* —we will not represent those edges labeled with *true*). The second stage performs (global) analysis [3] to gather information regarding, e.g., variable aliasing, groundness, side effects, etc. in order to remove edges from the dependency graph or to simplify the conditions labeling these edges, if they cannot be evaluated statically to completion. Labeled edges will result in run-time checks if conditional parallel expressions are allowed. Alternatively, unresolved dependencies can be assumed to always hold, and parallel execution will be allowed only between literals which have been statically determined to be independent. This approach saves run-time checks at the expense of losing some parallelism. Finally, the third stage transforms the original program into a parallel version by *annotating* it with parallel execution operators using the information gathered by the analyzers [22]. This annotation should respect the dependencies found in the original program while, at the same time, exploiting as much parallelism as possible.

This annotation process is the focus of this paper. We will present and evaluate new annotation algorithms which target and-parallelism primitives which can express richer dependency graphs than those which can be encoded with the *nested fork-join* approaches which have been previously proposed (e.g., [22]). Our hope is that since the transformed programs will contain in some cases more parallelism, we will be able to obtain better speedups for such cases.

2 Background and Motivation

We will introduce, with the help of an example, the well-known `&/2` operator for parallelism and its limitations, and we will show how better annotations for parallelism are possible when other, simpler primitives, are used.

2.1 *Fork-Join*-Style Parallelization

We will use as running example the following clause:

```
p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z).
```

and will assume that the dependencies detected between the literals in the predicate are defined by the graph $G = (V, E)$, shown in Figure 1. The vertices V correspond to the literals of the clause and there exists an edge between two literals L_i and L_j in E if $ind(L_i, L_j) \neq true$ (i.e., the literals

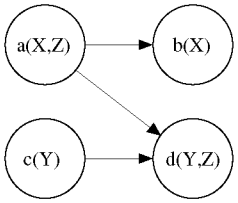


Fig. 1. Dependency graph for `p/3`.

```
p(X, Y, Z):-
        (a(X, Z), b(X)) & c(Y),
        d(Y, Z).
```

(a) *fj1*: Order-preserving

```
p(X, Y, Z):-
        a(X, Z) & c(Y),
        b(X) & d(Y, Z).
```

(b) *fj2*: Non-order-preserving

Fig. 2. *Fork-Join* annotations for `p/3` (Section 2).

L_i and L_j are dependent and thus the literal L_i has to be completed before the literal L_j), where *ind* is the notion of independence. As mentioned before, this information is obtained in our case from global data-flow analysis [3].

We will assume in the rest of the paper that all the dependencies are unconditional —i.e., conditional dependencies are assumed to be always false. This brings simplicity and avoids potentially costly run-time checks in the parallelized code at the expense of having fewer opportunities for parallelism. However, it has been experimentally found to be a good compromise [22, 3].

Conjunctive parallel execution has traditionally been denoted using the `&/2` operator instead of the sequential comma (','). The former binds more tightly than the latter. Thus, the expression "`a, b & c, d`" means that literals `b` and `c` can be safely executed in parallel after the execution of literal `a` finishes. When both `b` and `c` have successfully finished, execution continues with `d`.

While this single operator is enough to parallelize many programs, the class of dependencies it can express directly (i.e., dependency graphs with a nested fork-join structure) is a subset of that which can possibly appear in a program [22]. This makes parallelism opportunities to be inevitably lost in cases with a complex enough structure (e.g., that in Figure 1). Likewise, inter-procedural parallelism (i.e., parallel conjunctions which span literals in different predicates) cannot be exploited without program transformation.

In general, several annotations are possible for a given clause. As an example, Figure 2 shows two annotations for our running example.[3] Some goals appear switched w.r.t. their order in the sequential clause. This respects the dependencies in Figure 1, which reflects a valid notion of parallelism (i.e., if solution order is not important). If additional ordering requirements are needed (due to, e.g., side effects or impurity), these should appear as additional edges in the graph.

Note that none of the annotations in Figure 2 fully exploits all parallelism available in Figure 1: Figure 2(a) misses the parallelism between `b(X)` and `d(Y, Z)`, and Figure 2(b) misses the parallelism between `b(X)` and `c(Y)`.

One relevant question is which of these two parallelizations is better. Arguably, a meaningful measure of their quality is how long each of them takes to execute. We will term those times T_{fj1} and T_{fj2} for Figures 2(a) and 2(b), respectively. This length depends on the execution times of the goals involved (i.e., T_a, T_b, T_c, T_d), which we assume to be non-zero. T_{fj1} and T_{fj2} are:

$$T_{fj1} = \max(T_a + T_b,\ T_c) + T_d \tag{1}$$

[3] The parallelization `p :- a(X, Z), b(X) & c(Y), d(Y, Z)` has been left out of Figure 2. It would not add anything to the discussion as it would not change the comparison we make in Section 2.2.

$$T_{fj2} = \max(T_a,\ T_c) + \max(T_b,\ T_d) \tag{2}$$

Comparing the quality of the annotations in Figure 2(a) and Figure 2(b) boils down to finding out whether it is possible to show that $T_{fj1} < T_{fj2}$ or the other way around. It turns out that they are non-comparable. In fact:

- $T_{fj1} < T_{fj2}$ holds if, for example, $T_a + T_b < T_c$, $T_d < T_b$, and then $T_{fj2} = T_b + T_c$, $T_{fj1} = T_d + T_c$, and
- $T_{fj2} < T_{fj1}$ holds if, for example, $T_c \leq T_a$, $T_d \leq T_b$, and then $T_{fj1} = T_a + T_b + T_d$, $T_{fj2} = T_a + T_b$.

Several annotation algorithms have been proposed so far [22, 4] which use the `&/2` operator as the basic construction to express parallelism between goals. These annotators produce clauses that are parallelized differently, such as those in Figure 2. It is in principle possible to statically decide (or, at least, approximate) whether some annotation is better than some other, for example by using the number of goals annotated for parallelism in a clause or, more interestingly, by using information regarding the expected runtime of goals (see, e.g., [21, 19] and its references). However, finding an optimal solution is a computationally expensive combinatorial problem [22] and, in practice, annotators use heuristics which may be more or less appropriate in concrete cases.

2.2 Parallelization with Finer Goal-Level Operators

It has been observed [4, 5] that more basic constructions can be used to represent and-parallelism by using two operators, `&>/2` and `<&/1`, defined as follows:

Definition 1. `G &> H` *schedules goal* `G` *for parallel execution and continues executing the code after* `G &>H`. `H` *is a* handler *which contains (or* points to*) the state of goal* `G`.

Definition 2. `H <&` *waits for the goal associated with* `H` *to finish. After that point any bindings made by* `G` *are available to the executing thread.*

With the previous definitions, the `&/2` operator can be written as `A & B :- A &> H, call(B), H <&`. This indicates that any parallelization performed using `&/2` can be made using `&>/2` and `<&/1` without loss of parallelism. We will term these operators *dep-operators* henceforth.

Two motivations justify the use of these operators instead of `&/2`. Firstly, their implementation is (in our experience) actually easier to devise and maintain than the monolithic `&/2` [8], and, secondly, the dep-operators allow more freedom to the annotator (and to the programmer, if parallel code is written by hand) to

```
p(X, Y, Z) :-
        c(Y) &> Hc,
        a(X, Z),
        b(X) &> Hb,
        Hc <&,
        d(Y, Z),
        Hb <&.
```

Fig. 3. dep-operator-annotated clause

express data dependencies and, therefore, to extract more potential parallelism. We will now illustrate this last point (the former is out of our current scope).

Figure 3 shows an annotation of our running example using dep-operators. Note that this code allows executing in parallel `a/2` with `c/1`, `b/2` with `c/1`, and `b/1` with `d/2`. The execution time of `p/3`, based on that of the individual goals, is:[4]

$$T_{dep} = \max(T_a + T_b,\ T_d + \max(T_a,\ T_c)) \tag{3}$$

If we compare expression (3) with expressions (1) and (2), it turns out that:

- It is possible that $T_{dep} < T_{fj1}$, $T_{dep} < T_{fj2}$, $T_{dep} = T_{fj1}$, and $T_{dep} = T_{fj2}$ (possibly with different lengths for every goal in each case [7]).
- It is **not** possible that $T_{dep} > T_{fj1}$ or that $T_{dep} > T_{fj2}$.

This means that the annotation in Figure 3 cannot be worse than those of Figure 2, and can perform better in some cases. It is, therefore, a better option than any of the others.

In addition to these basic operators, other specialized versions can be defined and implemented in order to increase performance by adapting better to some particular cases. In particular, it appears interesting to introduce variants for the very relevant and frequent case of deterministic goals. For this purpose we propose two new operators: `&!>/2` and `<&!/1`. These specialized versions do not perform backtracking and do not prepare the execution data structures to cope with that possibility, which has previously been shown to result in a significant efficiency increase in the underlying machinery [23].

3 The UOUDG and UUDG Algorithms

In this section we will present two concrete algorithms which generate code annotated for unrestricted independent and-parallelism (as in Figure 3) starting from sequential code. The proposed algorithms process one clause at a time and work on a directed acyclic dependency graph (V, E), where nodes are associated with body goals in the clause. We require that literals which are lexically identical give rise to different nodes, by, e.g., attaching a unique identifier to them. This is necessary in order not to lose information when building sets of nodes.

We assume a preprocessing stage in each iteration of the algorithms which collapses sequences of mutually dependent goals into a single goal in the graph[5], i.e., $(\forall v_i, v_j \in Gr,\ v_i \rightsquigarrow v_j \ \vee\ v_j \rightsquigarrow v_i)$ and $(\forall (v_k, v_l) \in E,\ v_k \notin Gr \Rightarrow v_l \notin Gr)$, where Gr represents the sequence of goals and $x \rightsquigarrow y$ informs that there exists a path between the nodes x and y. For example, in `p:- a(X), b(X), c(X), d(Y), e(Y), f(X, Y)` the sets {`a/1, b/1, c/1`} and {`d/1,e/1`} are sequences in the clause, but they have a single outgoing dependency on `f/2`. Every one of these sequences can, for efficiency reasons, be folded into a unique predicate in order to avoid meta-interpretation of sequential conjunctions.

[4] See [7] for a deduction.

[5] In the case of the UOUDG algorithm, those goals must be consecutive in the original clause in order to preserve the order of the solutions.

Algorithm: UOUDG(G, Pub)

Input : **(1)** A directed acyclic graph $G = (V, E)$.
(2) A set of already forked goals.

Output: A clause parallelized in *unrestricted and* fashion in which the order of the solutions in the original clause is preserved.

begin
  if $V = \emptyset$ **then return** (true)
  else
    $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
    $Dep \leftarrow \{(v, I_v) \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
    if $Dep = \emptyset$ **then**
      $(pvt, Join) \leftarrow (u, V)$ s.t. $\forall (w \in (V \setminus \{u\}))\ .\ w \prec u$;
    else
      $(pvt, Join) \leftarrow$
        (u, S) s.t. $(u, S) \in Dep \wedge \forall ((w, D) \in (Dep \setminus \{(u, S)\}))\ .\ u \prec w$;
    end
    $Seq \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \rightarrow pvt \in E,\ v = pred(pvt)\}$;
    $Fork \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \prec pvt\} \setminus Seq$;
    $Join \leftarrow Join \setminus Seq$;
    $Pub \leftarrow Pub \cup Fork \cup Seq$;
    $G \leftarrow G - (Join \cup Seq)$;
    return (gen_body($Fork$, Seq, $Join$, $\emptyset$), UOUDG(G, Pub));
  end
end

Algorithm 1: UOUDG annotation algorithm.

The idea behind these algorithms is to publish goals for parallel execution as soon as possible and to delay issuing joins as much as possible —but always respecting the dependencies in the graph (as in Figure 1). Intuitively, this should maximize the number of goals available for parallel execution. In the following, both algorithms use an auxiliary definition to denote the set of nodes which are connected to some node v: $\texttt{incoming}(\texttt{v}, \texttt{E}) = \{u \mid (u \rightarrow v) \in E\}$.

Note that, as mentioned in Section 2.1, we will consider in this paper only unconditional parallelism. However, the algorithms that we describe can be adapted to deal with conditional parallelism without too much effort.

3.1 Order-Preserving Annotation: the UOUDG Algorithm

Algorithm 1 parallelizes a clause while preserving the order of the solutions by respecting the relative order of literals in the original clause. In order to keep track of that order, we assume that there is a relation $\prec$ on the literals L_i of the body of every clause H :- $L_1, L_2, \ldots, L_{k-1}, L_k$ such that $L_i \prec L_j$ iff $i < j$. Additionally, we assume that there is a partial function $pred$ defined as $pred(L_{i+1}) = L_i$, i.e., the literal at the left of some other literal in a clause. We assume $\prec$ and $pred$ are suitably extended to the nodes of the graph.[6]

[6] Note, also, that the graph edges must respect the $\prec$ relation: $(u \rightarrow v) \in E \Rightarrow u \prec v$. The graph would have been incorrectly generated otherwise.

At every recursion step, new nodes (i.e., literals) in the graph are selected to be published, joined, and executed sequentially. Subsequent iterations proceed with a simplified graph in which the literals which have been joined and executed sequentially, together with their outgoing edges, have been removed. The set of goals which have already been published is kept in a separate argument to schedule goals for parallel execution only once.

Two sets are key in each iteration: *Indep*, which contains the *sources* (i.e., all vertices without incoming edges in the current graph, which can therefore be published), and *Dep*, which contains tuples (v, I_v) where, for each non-source vertex v which can be reached from source vertices only, I_v is the set of source vertices ($I_v \subseteq Indep$) on which v depends. I.e., I_v is the set of vertices to be joined before v can start.

Also, *pvt* is the *pivot* vertex which will be used to decide which nodes are to be joined, taking into account that we do not want to change the order of solutions. If there are no *Dep* nodes, then all the remaining literals are already independent and we can join up to the rightmost literal in the clause. Otherwise, we select the leftmost node among those which have dependencies which can be fulfilled in one step. These dependencies are readily available in *Dep*. Note that as we select the leftmost node among those which can be joined, we are delaying as much as possible joining nodes —or, alternatively, we are performing in every step only the joins which are needed to continue one more step. This is aimed at maximizing the number of parallel goals being executed at any moment.

It is possible for a literal to be scheduled to be forked and then immediately joined. In order to detect these situations, which in practice would cause unnecessary overhead, we select (in Seq) the literal (only one) to which this applies, and it is not taken into account for the set of $Fork$ed literals and removed from the set of the $Join$ed literals.

The algorithm then continues outputting a parallelized expression (returned by `gen_body`, Algorithm 3) composed with the parallelization of a simplified graph, generated by a recursive call. Algorithm 3 is able to use determinism information to reorder goals. Since Algorithm 1 preserves the order of solutions, we do not use this capability at the moment. Therefore an empty set is passed as determinism data and we define the function $det(Lit, DetInfo)$ (used by Algorithm 3) to return *false* if $DetInfo = \emptyset$, thus safely assuming non-determinism.

Termination can be proved based on the following observation: G is a finite graph and it is simplified in each iteration provided $Join$ or Seq are non-empty. But $Join$ is always non-empty because it is either V (which is non-empty) when $Dep = \emptyset$ or else it is the second component of a tuple in Dep when $Dep \neq \emptyset$, and this component is by definition non-empty. Note that we are not using acyclicity to prove termination. However, all input graphs will be acyclic by definition.

3.2 Non Order-Preserving Annotation: the UUDG Algorithm

Algorithm 2 follows the same idea underlying Algorithm 1: publish early and join late. However, it has more freedom to publish goals, since the order of solutions

Algorithm: UUDG(G, Pub, I_D)

Input : **(1)** A directed acyclic graph $G = (V, E)$. **(2)** A set of goals already forked. **(3)** Determinacy information.

Output: An unrestricted parallelized clause in which the order of the solutions in the original clause needs not be preserved.

```
begin
    if V = ∅ then return (true);
    else
        Indep ← {v | v ∈ V, incoming(v, E) = ∅};
        Dep ← {I_v | v ∈ V, I_v = incoming(v, E), I_v ≠ ∅, I_v ⊆ Indep};
        if Dep = ∅ then
            SS ← ∅;
            Join ← V;
        else
            SS ← {I | I ∈ Dep, |I| = min_card(Dep)};
            Join ← s s.t. s ∈ SS ;            /* s any element from SS */
        end
        if (Join ∩ (Indep \ Pub)) = ∅ then
            Seq ← ∅;
        else
            Seq ← {v} s.t. v ∈ (Join ∩ (Indep \ Pub)) ;   /* v any element */
        end
        Fork ← Indep \ (Pub ∪ Seq);
        Join ← Join \ Seq;
        Pub ← Pub ∪ Fork ∪ Seq;
        G ← G−(Join ∪ Seq);
        return (gen_body(Fork, Seq, Join, I_D), UUDG(G, Pub, I_D));
    end
end
```

Algorithm 2: UUDG annotation algorithm.

does not need to be preserved. This is implemented by selecting, among the sets of goals which can be joined at every moment, the one with the lowest cardinality —i.e., we join as few goals as possible, thus postponing the rest of the joins as much as possible, in order to exploit more parallelism. This is taken care of by $\mathtt{min\_card}(\mathtt{S}) = \min(\{|s| \mid s \in S\}$, which returns the size of the smallest set in S.

Note that a random selection from a set is done at two points. Data regarding, e.g., the relative run time of goals would allow us to take a more informed decision and therefore precompute a perhaps better scheduling. Since we are not using this information here, we just pick any available goal to join / execute sequentially.

Algorithm 2 again uses Algorithm 3 to output a parallelized clause. In this case Algorithm 3 makes use of determinism information as follows:

- Since we already have the possibility of switching goals around, we try to minimize relaunching goals which are likely to be executed in parallel by forking deterministic goals first.

Algorithm: gen_body(*Fork*, *Seq*, *Join*, I_D)

Input : **(1)** A set of vertices to be forked. **(2)** A set of vertices to be sequentialized. **(3)** A set of vertices to be joined. **(4)** Determinacy information.

Output: A parallelized sequence of literals *Exp*.

begin

$Exp \leftarrow$ (`true`);
$ForkDet \leftarrow \{g \mid g \in Fork, det(g, I_D)\}$;
$ForkNonDet \leftarrow \{g \mid g \in Fork, \neg det(g, I_D)\}$;
$JoinDet \leftarrow \{g \mid g \in Join, det(g, I_D)\}$;
$JoinNonDet \leftarrow \{g \mid g \in Join, \neg det(g, I_D)\}$;
forall $v_i \in ForkDet$ **do** $Exp \leftarrow (Exp,\ v_i$ `&!>` $H_{v_i})$;
forall $v_i \in ForkNonDet$ **do** $Exp \leftarrow (Exp,\ v_i$ `&>` $H_{v_i})$;
if $Seq = \{v\}$ **then** $Exp \leftarrow (Exp,\ v)$;
forall $v_i \in JoinDet$ **do** $Exp \leftarrow (Exp,\ H_{v_i}$ `<&!`$)$;
forall $v_i \in JoinNonDet$ **do** $Exp \leftarrow (Exp,\ H_{v_i}$ `<&`$)$;
return *Exp*;

end

Algorithm 3: Determinism-aware generation of a parallel body.

G=(V,E)	I	D	J	S	F	J\S	P	Parallel Code
$(\{a,b,c,d\},\{(a,b),(a,d),(c,d)\})$							$\emptyset$	`p(X,Y,Z) :-`
$(\{a,b,c,d\},\{(a,b),(a,d),(c,d)\})$	$\{a,c\}$	$\{b,d\}$	$\{a\}$	$\{a\}$	$\{c\}$	$\emptyset$	$\{a,c\}$	`c(Y) &> Hc, a(X,Z),`
$(\{b,c,d\},\{(c,d)\})$	$\{b,c\}$	$\{d\}$	$\{c\}$	$\emptyset$	$\{b\}$	$\{c\}$	$\{a,b,c\}$	`b(X) &> Hb, Hc <&,`
$(\{b,d\},\emptyset)$	$\{b,d\}$	$\emptyset$	$\{b,d\}$	$\{d\}$	$\emptyset$	$\{b\}$	$\{a,b,c,d\}$	`d(Y,Z), Hb <&.`
$(\emptyset,\emptyset)$								

Table 1. Iterations of the UUDG algorithm when parallelizing `p/3`.

– Additionally, when a goal is known to have exactly one solution, we can use specialized versions of the dep-operators [8] which do not need to perform bookkeeping for backtracking (always complex in parallel implementations), and are thus more efficient.

This program information can often be automatically inferred by the abstract interpretation-based determinism analyzer in CiaoPP [18], and is provided as input to the proposed annotators. Alternatively, this information can be stated by the programmer via assertions [13].

Example 1 (UUDG Annotation). In order to illustrate how the UUDG algorithm works, Table 1 shows the results obtained at each of the iterations of the parallelization process for the `p/3` predicate introduced in Section 2.1 and whose dependency graph is shown in Figure 1. Columns are labeled with the first character of each of the variables they represent. Note that in the first algorithm step, both a and c are candidates for parallel execution (they are in *Indep*). However, as a has to be joined too (it is necessary to continue executing either b or d) it is selected to be sequentially executed.

AIAKL	An abstract interpreter for the AKL language.
FFT	An implementation of the Fast Fourier transform.
FibFun	A version of **Fib** written in functional notation.
Hamming	A program to compute the first N Hamming numbers.
Hanoi	A program to compute movements to solve the well-known puzzle.
Takeuchi	Computes the Takeuchi function.
WMS2	A scheduler assigning a number of workers to a series of jobs.

Table 2. Benchmark programs

4 Performance Evaluation

Our annotation algorithms have been integrated in the Ciao/CiaoPP system [13]. Information gathered by the analyzers on variable sharing, groundness, and freeness is used to determine goal independence, using the libraries available in CiaoPP. Determinism is used in the annotators as described previously.

As execution platform we have used a high level implementation of the proposed parallelism primitives [8], which we have developed as an extension of the Ciao system. This implementation is an evolution and simplification of [12] which is based on raising the level of certain components to the level of the source language and keeping only some selected operations (related to thread handling, locking, etc.) at a lower level. This approach does not eliminate altogether modifications to the abstract machine, but it greatly simplifies them. It should be noted however that the dep-operators do not assume any particular architecture: while our current implementation and all the performance results were obtained on a multicore machine, the techniques presented can be also applied in distributed memory machines —and in fact, the first prototype implementation of the dep-operators [5, 4] was actually made on a distributed environment.

We have evaluated the impact of the different annotations on the execution time by running a series of benchmarks (briefly described in Table 2) in parallel. Table 3 shows the speedups obtained *with respect to the sequential execution*, i.e., they are *actual* speedups,[7] when using from 1 to 8 threads. The machine we used is a Sun UltraSparc T2000 (a *Niagara*) with 8 4-thread cores.[8] The *fork-join* annotators we chose to compare with are MEL [22] (which preserves goal order and tries to maximize the length of the parallel expressions) and UDG [4] (which can reorder goals). MEL can add runtime checks to decide dynamically whether to execute or not in parallel. In order to make the annotation unconditional (as the rest of the annotators we are dealing with), we simply removed the conditional parallelism in the places where it was not being exploited. This is why it appears in Table 3 under the name *UMEL*.

All the benchmarks executed were parallelized automatically by CiaoPP, starting from their sequential code. Since UOUDG and UUDG can improve the results of fork-join annotators only when the code to parallelize has at least a cer-

[7] This is the reason why some speedups start below 1 for, e.g., one thread.

[8] We did not use more than 8 cores since in that case, and due to access to shared units, speedups are sublinear even for completely independent tasks.

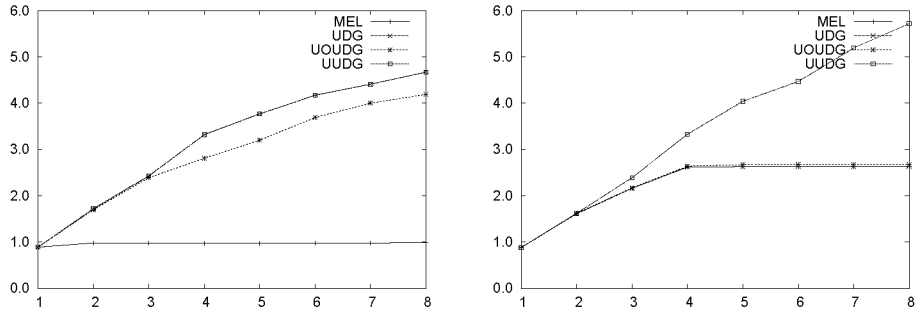
Benchmark	Annotator	Number of threads							
		1	2	3	4	5	6	7	8
AIAKL	UMEL	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
	UOUDG	0.97	1.55	1.48	1.49	1.49	1.49	1.49	1.49
	UDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
	UUDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
FFT	UMEL	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UOUDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UUDG	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
FibFun	UMEL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UOUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
	UDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	UMEL	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UOUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
	UDG	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
Hanoi	UMEL	0.89	0.98	0.98	0.97	0.97	0.98	0.98	0.99
	UOUDG	0.89	1.70	2.39	2.81	3.20	3.69	4.00	4.19
	UDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
	UUDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
Takeuchi	UMEL	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UOUDG	0.88	1.62	2.17	2.64	2.67	2.67	2.67	2.67
	UDG	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UUDG	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
WMS2	UMEL	0.85	0.81	0.81	0.81	0.81	0.81	0.81	0.81
	UOUDG	0.99	1.09	1.09	1.09	1.09	1.09	1.09	1.09
	UDG	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	UUDG	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

Table 3. Speedups for several benchmarks and annotators.

tain level of complexity, not all benchmarks with (independent) parallelism can benefit from using the dep-operators. Additionally, comparing speedups obtained with programs parallelized using order-preserving and non-order-preserving annotators is not completely meaningful.

Note that in this paper we are not focusing on the speedups themselves. Although of utmost practical interest, raw speed is very connected with the implementation of the underlying parallel abstract machine, and improvements on it can be expected to uniformly affect all parallelized programs. Rather, our main focus of attention is in the *comparison* among the speedups obtained using different annotators.

A first examination of the experimental results in Table 3 allows inferring that in no case is UUDG worse than any other annotator, and in no case is UOUDG worse than (U)MEL. They should therefore be the *annotators of choice* if available. Besides, there are cases where UOUDG is better than UDG, and the



(a) Hanoi (b) Takeuchi

Fig. 4. Speedups with different annotations for Hanoi and Takeuchi.

other way around, which is in accordance with the non-comparable nature of these two algorithms.

Among the cases in which a better speedup is obtained by some of the U(O)UDG annotators, improvements range between "no improvement" (because no benefit is obtained for some particular cases and combinations of annotators) to an increase of 757% in speedup, with several other stages in between. Also, it is worth pointing out that the speedup does not stabilize in any benchmark (at least in a sizable amount) as the number of threads increases; moreover, in some cases the difference in speedup between the restricted and the unrestricted versions grows substantially with the number of threads. This can (clearly) be seen in, e.g., Figure 4(b).

Finally, we would like to comment specially on three benchmarks. **FibFun** is the result of parallelizing a definition of the Fibonacci numbers written using the functional notation capabilities of Ciao [6]. Because of the order in which code is generated in the (automatic) translation into Prolog, the result is only parallelizable by UOUDG and UUDG, hence the speedup obtained in this case. The case of **Hanoi** is also interesting, as it is the first example in [22]: in the arena of order-preserving parallelizers, UOUDG can extract more parallelism than MEL for this benchmark. Lastly, the **Takeuchi** benchmark has a relatively small loop which only allows parallelizing with a simple `&/2`. However, by unrolling one iteration the resulting body has dependencies which are complex enough to take advantage of the increased flexibility of the dep-operator annotators.

5 Conclusions

We have proposed two annotation algorithms which perform a source-to-source transformation of a logic program into an unrestricted independent and-parallel version of itself. Both algorithms rely on the use of more basic high-level primitives than the fork-join operator, and differ on whether the order of the solutions in the original program must be preserved or not. We have implemented the proposed algorithms in the CiaoPP system, which infers automatically groundness,

sharing, and determinacy information, used to simplify the initial dependency graph. The results of the experiments performed show that, although the parallelization provided by the new annotation algorithms is the same in quite a few of the traditional parallel benchmarks, it is never worse and in some cases it is significantly better. This supports the observations made based on the expected performance of the annotations. We have also noticed that the benefits are larger for programs with high numbers of goals in their clauses, since more complex graphs make the ability to exploit unrestricted parallelism more relevant.

References

1. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
2. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.
3. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM TOPLAS*, 21(2):189–238, March 1999.
4. D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.
5. D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.
6. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.
7. A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. Technical Report CLIP5/2007.0, Technical University of Madrid (UPM), School of Computer Science, UPM, June 2007.
8. A. Casas, M. Carro, and M. Hermenegildo. Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*. Springer-Verlag, January 2008.

9. G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.
10. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, July 2001.
11. M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14):1685–1708, December 2000.
12. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
13. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
14. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
15. M. Hermenegildo and R. Warren. Designing a High-Performance Parallel Logic Programming System. *Computer Architecture News, Special Issue on Parallel Symbolic Programming*, 15(1):43–53, March 1987.
16. Sverker Janson. *AKL. A Multiparadigm Programming Language.* PhD thesis, Uppsala University, 1994.
17. A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.
18. P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.
19. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.
20. E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
21. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.
22. K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.
23. E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *The Computer Languages Journal*, 22(2/3):115–142, July 1996.
24. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.
25. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.